FIX PROTOCOL
INDUSTRY-DRIVEN MESSAGING STANDARD℠

Implementation & Optimization Working Group Journal

April 2009

Volume 1 Issue 1

DISCLAIMER

Table of Contents

Contributors

The following people submitted articles in the preparation of this white paper:

- Clive Browning, Rapid Addition
- Mahesh Kumaraguru
- Cooper Noone, Counterparty Systems
- Ryan Pierce, FIX Protocol Ltd.
- John Prewett, Citi
- Qingjun Wei, Teraspaces

Several other people should be credited who have been involved in reviewing and coordination, especially:

- Kathleen Callahan, FIX Protocol Ltd

Articles

This white paper contains articles on the following subjects:

- FIX C# usage tips
- Java Performance – Avoiding Garbage Collection
- Resilience
- FIX Java Type Safety
- Disabling The Nagle Algorithm
- I/O Models and I/O Completion Ports

Introduction: Building the Case for Cooperation

By Ryan Pierce, FIX Protocol Ltd., and John Prewett, Citi,
Co-chairs, FPL Implementation and Optimization Working Group

The Implementation and Optimization Working Group had its beginnings at the FPL Americas Conference in November, 2008. John Prewett was speaking on a panel about optimizing FIX implementations. Ryan Pierce was sitting in the audience. John was talking about the Nagle algorithm, an insidious little quirk enabled by default in most operating systems that will wreak havoc on FIX performance if not disabled. Ryan already knew about Nagle, but such knowledge was learned the hard way as he tried to discover why a perfectly good FIX implementation randomly took several hundred millisecond coffee breaks, an eternity to algorithmic traders. One line of code was all it took to disable Nagle, and the problem immediately went away.

How many people still don't know about this? Why isn't this documented somewhere? The need for a way to capture this kind of knowledge was apparent, and an e-mail requesting the formation of an Implementation and Optimization Working Group left Ryan's BlackBerry before the panelists stepped down from the podium.

Historically, implementation issues have been something of a blind spot for FIX Protocol Ltd., and for good reasons. FPL writes and maintains a protocol. Vendors and firms in the financial industry implement the protocol. FPL's job is to define the representation of data that goes over a wire. How to identify a security, or how to respond to a sequence number greater than expected, are issues solidly within FPL's domain. But issues such as disabling the Nagle algorithm or avoiding C# garbage collection don't affect the FIX specification at all, and so, outside of recruiting experts to speak at conferences, FPL has not made a concerted effort to document technical best practices surrounding implementation and optimization.

Likewise, the community of vendors can be fiercely competitive. If one FIX engine or FIX-enabled EMS performs better than a competitor's product, that is a competitive advantage that can drive sales.

But everyone who has invested heavily in FIX faces a common enemy, or rather, three common enemies. Fear. Uncertainty. And Doubt.

The Tag=Value syntax of the FIX Protocol is, admittedly, a compromise. It is predominantly an ASCII-based standard. It is about as compact as one can expect for a protocol based around simple encoding rules that can still be read and understood readily by a human.

And so some people evaluating the FIX Protocol jump to the conclusion that FIX must be slow and inefficient. Often, they use this to justify proprietary protocols which may be marginally more compact, but which come at a terrific cost. Often, the incredible extensibility inherent in FIX is lost. And most proprietary protocols don't have the compatibility with thousands of firms and vendor products that is inherent in FIX. The cost of implementation, QA, and deployment required to implement proprietary protocols is high, especially in these current economic conditions where IT budgets are slashed industry-wide.

These days, firms are reporting order ack times that are a few hundred microseconds. Firms are advertising throughput of many tens of thousands of FIX messages per second parsed on FIX engines running on inexpensive PC hardware. In most cases, inefficiencies in Tag=Value FIX can't explain perceived latency. FIX becomes the excuse; the real cause often is a poor or inadequate implementation.

Now is the time for open cooperation in improving the performance of FIX systems across the industry. Having the fastest FIX engine ceases to be a competitive advantage when an exchange wrongly concludes that FIX itself is slow and adopts a proprietary protocol. Fortunately, firms are recognizing the need for frank discussions and sharing of knowledge across the industry.

This is the role that FPL's Implementation and Optimization Working Group is meant to fill. We provide an open, neutral forum where firms can discuss and share best practices.

Currently, the primary scope of our work is the FIX Tag=Value syntax. Clearly, the FAST Protocol representation of FIX outperforms Tag=Value, and FAST dominates FIX market data distribution. But, as of this issue's publication, FAST has seen little adoption for order routing, where Tag=Value syntax still holds ground.

We recognize that best practices for implementation and optimization are at the cutting edge of financial IT research. New technologies and techniques are emerging rapidly, such that best practices a year ago might be very different from best practices today. We don't expect all the experts to agree on one "best" way to implement FIX.

The Working Group has decided to publish a peer-reviewed journal, similar to those published by the academic and scientific communities. Each article presents best practices according to the opinion of its respective author. These are not "standards" nor does FPL or the Implementation and Optimization Working Group endorse them as true and correct. Rather, our goal is to collect and publish opinions from a diverse group of experts, who may not always agree with each other.

We welcome additional involvement in the Implementation and Optimization Working Group. Submissions of articles, or responses to articles, are encouraged. We also welcome suggestions for articles, as well as those who wish to help with peer review. Anyone interested should contact fpl@fixprotocol.org and request to be added to the Implementation and Optimization Working Group.

FIX C# Usage Tips

By Clive Browning, Rapid Addition

Overview

This document provides C# usage tips that have been brought to the surface during the development of the Rapid Addition FIX engine and the Rapid Addition FAST engine product suites. This is not an exhaustive list. The purpose of this document is to highlight specific areas within C# and .Net that can cause performance bottlenecks and to provide some guidelines in improving code performance in general.

What is Slow

- Instantiating an object is costly.
- Garbage collection.
- Runtime type casting.
- String concatenation using the String class.
- Building for debug adds runtime overheads.
- Nagle algorithm on the socket.

What is Fast

- Object pools reduce the impact of the garbage collector.
- Object pools can cut down on expensive object creation.
- Generics classes cut out runtime type checking.
- StringBuilder for string concatenation operations.
- Building for release for performance.

These topics will be investigated further in the following sections.

General Guidelines

The following general guidelines should be kept in mind whilst developing a system that has a low latency requirement. These guidelines are not exhaustive.

Garbage Collection and Object Pools

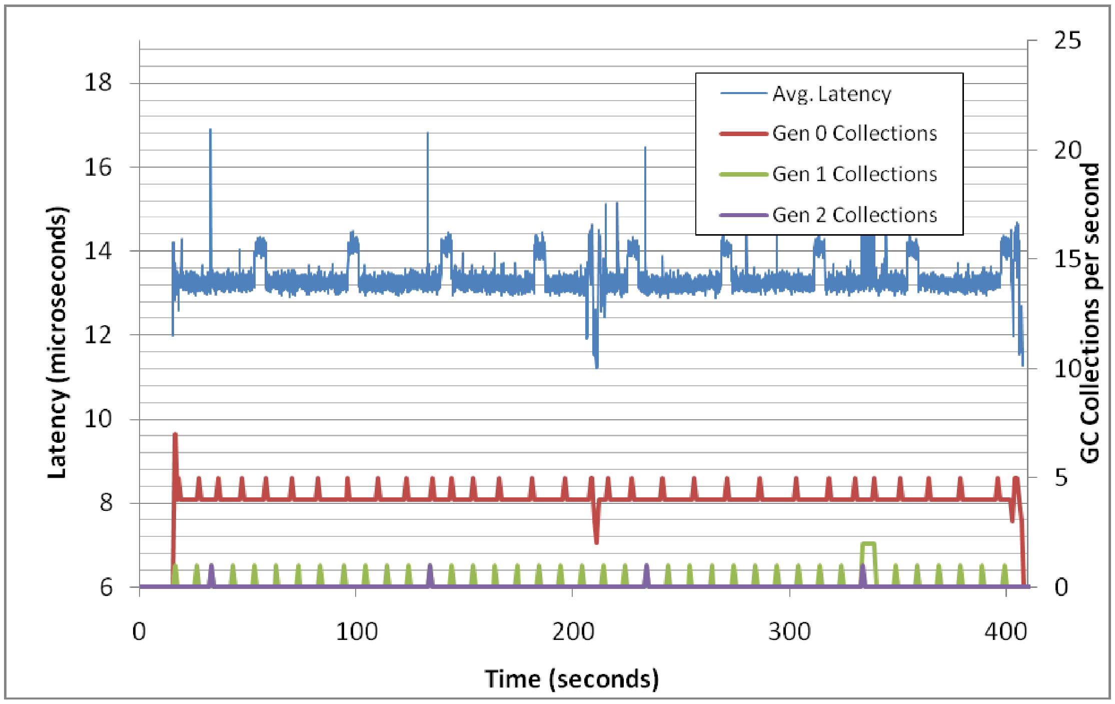
The .Net runtime is a managed environment that takes care of memory management. This is a real bonus in effectively removing many issues relating to mismatched calls to "new" and “delete” operator calls when allocating and releasing memory. In .Net there is no “delete” operator. The “delete” operator functionality is provided automatically by the .Net garbage collector.

There is an adverse side effect to this. The garbage collector has to take control and run memory management code. This is accomplished in the .Net runtime by having 3 generations of objects (Gen-0, Gen-1 and Gen-2). A heap based object is initially assigned to Gen-0 and progresses to Gen-1 if it lives beyond a Gen-0 collection and into Gen-2 if it lives beyond a Gen-1 collection.

Running a generation 0 collection is light-weight, a generation 1 collection is a little heavier, and a generation 2 collection effectively stalls the process whenever it kicks in. Currently generation 2 collections have to stop all threads so they can move objects around in memory and resolve object pointers to their new positions in memory.

Memory management is not an issue restricted to the .Net runtime. Unmanaged C / C++ applications also run into similar issues when their managed heap is exhausted. At that point the managed heap has to be re-organised so that small pockets of memory can be aggregated in order to fulfill a memory allocation request.

The following graph exhibits a piece of .Net 2 C# code running under load. The red line shows the background generation 0 collection that is carried out continuously. The green line shows the generation 1 collection that kicks in regularly. And the purple line shows the generation 2 collection that kicks in intermittently and has a serious impact on the latency (blue line). The spikes in the latency coincide with the generation 2 collection. Generation 2 collections are extremely costly and should be avoided if possible for low latency solutions.



To reduce the likelihood of a generation 1 or generation 2 collection (or in the case of the C/C++ to reduce the likelihood of a complete re-organisation of the managed heap) object pools can be utilised.

There are three core requirements for an object pool implementation: a factory to create the objects when required, the object pool and objects that support re-use (for example via a Reset method).

Interfaces

Interfaces are ideal for providing an architecture to support multiple implementations with differing requirements. For example, different implementations of a message store could be developed depending on the requirement for performance. If low-latency high performance is required a file or RAM disk based implementation can be developed. If a fully transactional based system is required then a database implementation can be developed.

Whilst the use of interfaces does not directly provide an improvement in performance, the pattern does allow for individual components to be replaced with more performant implementations without breaking the system.

In the example below, we have an IMessageStore interface which has been implemented by a FileMessageStore and a SqlMessageStore (not shown). Different Session class instances can be instantiated with the different message store options whilst making use of the same Session implementation.

Interface definition for the message store:

```
interface IMessageStore
{
    void Save();
    List<Message> Load(int StartSeqNo, int EndSeqNo);
}
```

Session object using the message store interface:

```
class Session
{
    public Session(IMessageStore messageStore);
}
```

Creating the session with the file based message store:

```
Session sessionUsingFileStore = new Session(new FileMessageStore());
```

Creating the session with the SQL based message store:

```
Session sessionUsingSqlStore = new Session(new SqlMessageStore());
```

TCP/IP Socket

Turn off the Nagle algorithm on the C# Socket object. The benefits of doing this are not realised under continuous heavy loads. But under most load situations this will ensure latency is kept low.

The Nagle algorithm is a means of improving the efficiency of TCP/IP based networks by reducing the number of packets that need to be sent over the network. However, this involves aggregating small messages and sending them in a single packet. This can result in a higher latency for messages that are smaller than the TCP/IP packet size sent at relatively large intervals of time.

Turning off the nagle algorithm setting is accomplished as follows when using the .Net Socket class:

```
static Socket CreateSocket()
{
    Socket fixSocket = new Socket();
    fixSocket.NoDelay = true;
    return fixSocket;
}
```

Consider increasing the Send and Receive socket buffer sizes to cope with bursts of activity that generate peaks of FIX messages traffic. The SendTimeout can be used when latency is of absolute importance and any messages not delivered within a critical time span should be catered for:

```
static void InitialiseSocketOptions(Socket fixSocket)
```

```
        {
            fixSocket.ReceiveBufferSize = 20 * 1024;
            fixSocket.SendBufferSize = 100 * 1024;
            fixSocket.SendTimeout = 10000;
        }
```

The default value for the ReceiveBufferSize is 12800 bytes and the default value for the SendBufferSize is 49152 bytes. The default setting for SendTimeout in milliseconds is 0, which means it will not timeout.

Resend Request Processing

Be aware of large resend-requests hitting a message store. Consider chunking the access to the message store to prevent numerous messages being loaded into memory. This also has the benefit that the I/O costs on the socket send can be intermingled with any I/O costs in retrieving data from the message store.

Consider caching the last "x" messages sent out by the FIX Engine so they are readily available for any resend-request operation. Resend requests normally only occur at login time during a resynchronisation process so do not affect the general performance of the FIX Engine. However, sessions dropping and re-establishing during the trading day can put extra pressure on the message store implementation.

Object Models

Object models that sit on top of the raw FIX message provide tremendous benefits in terms of usability for the coder. This ease of use can come at a penalty in terms of performance. For best performance it is often better to use the underlying raw FIX message implementation rather than code to an extra layer of abstraction.

Using base message class:

```
        Session session = new Session(new FileMessageStore());
        Message message = new Message("D");
        message.Add(55, "RA");
        session.Send(message);
```

Using type specific class:

```
        Session session = new Session(new FileMessageStore());
        NewOrderSingle newOrderSingle = new NewOrderSingle();
        newOrderSingle.Symbol = "RA";
        session.Send(newOrderSingle);
```

In the example above, using the base Message class is likely to provide better performance than using the specific NewOrderSingle class.

Data Type Conversions

Data arrivals on the socket as a byte array. This contains tag-values that need to be converted at some point for use within the application. These data type conversions are relatively expensive and can generate garbage, especially when using String objects as an interim container for the type conversion.

Consider developing type converters that convert direct from ASCII byte arrays to the primitive object types such as int, String, DateTime, and decimal, thus avoiding an intermediate string conversion.

Multi-Threading

The theory is that the performance of parsing a FIX message could be improved by using multiple threads on a multi-core or multi-processor system.

In practice, the overheads of creating, managing and synchronising multiple threads for the parsing process outweigh any benefits. The parsing process can be a fairly quick operation which may well be sub millisecond, whilst thread wake-up times can be measured in milliseconds.

Using Exception Handling Wisely

Like other languages that provide the ability to handle exceptions in a structured manner, C# and .Net have the same benefits and overheads. Exception handling should be used for processing exceptions and not for normal processing.

```
public static void ExceptionHandling()
{
    try
    {
        Session session = new Session(new FileMessageStore());
        session.Send(new Message());
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Prefer To Use Generics

Always make use of the .Net 2 Generics based classes. For example, use the Generics List<> class instead of the object based ArrayList class.

Working with non Generics array list:

```
static void WorkWithArrayList()
{
```

```
            ArrayList myArrayList = new ArrayList();
            myArrayList.Add("First item");
            string item = (string)myArrayList[0];
        }
```

Working with Generics list class:

```
        static void WorkWithList()
        {
            List<string> myList = new List<string>();
            myList.Add("First item");
            string item = myList[0];
        }
```

There are two main reasons for this:

First, using Generics ensures that the code is type-safe at compile time and therefore no casting is required.

Second, using Generics is more performant because no runtime type checking is required to ensure the casts are correct at runtime.

Examining the underlying MSIL (Microsoft Intermediate Language) for this code reveals that there is a runtime "castclass" call present when using ArrayList class that is not required for the Generics List class:

MSIL for ArrayList class:

```
  IL_0015:  callvirt   instance object
[mscorlib]System.Collections.ArrayList::get_Item(int32)
  IL_001a:  castclass  [mscorlib]System.String
  IL_001f:  stloc.1
  IL_0020:  ret
} // end of method Generics::WorkWithArrayList
```

MSIL for Generics List class:

```
  IL_0015:  callvirt   instance !0 class
[mscorlib]System.Collections.Generic.List`1<string>::get_Item(int32)
  IL_001a:  stloc.1
  IL_001b:  ret
} // end of method Objects::WorkWithList
```

There is extra runtime type checking required when accessing an item in the ArrayList class which is not required when using the Generics List<> class.

Debug vs Release Code

There is a performance benefit from running code compiled for release rather than for debug. The following MSIL snippet shows the differences in the compiled code:

The following C# code

```
        static void WorkWithList()
        {
            List<string> myList = new List<string>();
            myList.Add("First item");
            string item = myList[0];
        }
```

Generates the following MSIL when compiled for debug:

```
.method private hidebysig static void  WorkWithList() cil managed
{
  // Code size       28 (0x1c)
  .maxstack  2
  .locals init ([0] class [mscorlib]System.Collections.Generic.List`1<string> myList,
           [1] string item)
  IL_0000:  nop
  IL_0001:  newobj     instance void class
[mscorlib]System.Collections.Generic.List`1<string>::.ctor()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  ldstr      "First item"
  IL_000d:  callvirt   instance void class
[mscorlib]System.Collections.Generic.List`1<string>::Add(!0)
  IL_0012:  nop
  IL_0013:  ldloc.0
  IL_0014:  ldc.i4.0
  IL_0015:  callvirt   instance !0 class
[mscorlib]System.Collections.Generic.List`1<string>::get_Item(int32)
  IL_001a:  stloc.1
  IL_001b:  ret
} // end of method Objects::WorkWithList
```

And when compiled for release:

```
.method private hidebysig static void  WorkWithList() cil managed
{
  // Code size       26 (0x1a)
  .maxstack  2
  .locals init ([0] class [mscorlib]System.Collections.Generic.List`1<string> myList)
  IL_0000:  newobj     instance void class
[mscorlib]System.Collections.Generic.List`1<string>::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  ldstr      "First item"
  IL_000c:  callvirt   instance void class
[mscorlib]System.Collections.Generic.List`1<string>::Add(!0)
  IL_0011:  ldloc.0
  IL_0012:  ldc.i4.0
  IL_0013:  callvirt   instance !0 class
[mscorlib]System.Collections.Generic.List`1<string>::get_Item(int32)
  IL_0018:  pop
  IL_0019:  ret
} // end of method Objects::WorkWithList
```

So what’s the difference?

- Additional nop instructions added through the routine.
- And a stloc.1 rather than a pop at the end of the routine.

The nop instructions are there so that breakpoints can be inserted during debug sessions. Although it is a no-operation, it does take some finite processing time. A more complex method will result in more nop instructions being added.

The stloc.1 instruction rather than a pop instruction is perplexing; our best guess is so that in debug mode the return from the indexer on the List<> class can be inspected if required.

Optimisation Project Setting

The Visual Studio project properties has a check box setting marked as "optimize code". This is used to enable or disable optimizations performed by the compiler to make the output file smaller, faster, and more efficient. The exact nature of these optimisations we have not investigated fully.

Performance Counters

Use performance counters wisely, as there is an overhead in using them. Specifically, counters that measure send or receive times per FIX message can be very costly because a high performance timer is required to generate timing data on the order of microseconds. For example, the .Net StopWatch object provides the processor Frequency and a ElapsedTicks property so that elapsed times can be calculated in microseconds. This calculation is relatively heavy weight:

```
Stopwatch stopWatch = new Stopwatch();

stopWatch.Start();

long timeInMilliseconds = stopWatch.ElapsedMilliseconds;

long timeInMicroSeconds = (1000000 * ((double)stopWatch.ElapsedTicks /
                                       (double)Stopwatch.Frequency));
```

For milliseconds no expensive conversions or calculations are required. However, for microseconds an expensive calculation is required.

Foreach vs For

Is there any overhead in using "foreach" over a traditional "for" loop? The answer is it depends on what you are iterating over. Consider the following examples using non Generics and Generics:

Non Generics code:

```
ArrayList myArrayList = new ArrayList();
```

```
        myArrayList.Add(22);
        myArrayList.Add("A string");
        myArrayList.Add(DateTime.Now);

        foreach (string item in myArrayList)
        {
            Console.WriteLine(item);
        }
```

Generics code:

```
        List<string> myList = new List<string>();

        myList.Add("First");
        myList.Add("Second");

        foreach (string item in myArrayList)
        {
            Console.WriteLine(item);
        }
```

There is runtime type checking occurring in the "foreach" based loop that is not obvious from looking at the C# code. In fact, the non Generics code sample will throw an exception when an item in myArrayList is encountered and it is not of type String.

This problem is immediately apparent at compile time if a traditional "for" loop had been used in the non Generics code because a compile time type conversion error would have resulted:

```
        for (int index = 0; index < myArrayList.Count; index++)
        {
            string item = myArrayList[index];
            Console.WriteLine(item);
        }
```

The answer really is to use Generics.

String and StringBuilder

The String object is immutable. This is great. And this is also bad. It is good because multiple instances of String objects can re-use the same underlying immutable memory representation of the string data. However, it is bad because any changes that need to be applied to the String have to be made to a copy of the String. So even replacing a single character within a String object results in a new String object being created:

When constructing Strings, use the StringBuilder obect. The following code snippets demonstrate the less performant method of using String concatenation and the more performant method using the StringBuilder and the Append method.

Constructing a FIX message using a String:

```
        static string ConvertToString1(FixMessage fixMessage)
        {
            string fixMessageAsString = "";
            foreach (FixTagValue tagValue in fixMessage.TagValues)
            {
                fixMessageAsString +=
                              tagValue.Tag.ToString() +
                              EQUALS + tagValue.Value + SOH;
            }

            return fixMessageAsString;
        }
```

Constructing a FIX message using a StringBuilder:

```
        static string ConvertToString2(FixMessage fixMessage)
        {
            StringBuilder fixMessageAsStringBuilder = new StringBuilder();

            foreach (FixTagValue tagValue in fixMessage.TagValues)
            {
                fixMessageAsStringBuilder.Append(tagValue.Tag);
                fixMessageAsStringBuilder.Append(EQUALS);
                fixMessageAsStringBuilder.Append(tagValue.Value);
                fixMessageAsStringBuilder.Append(SOH);
            }

            return fixMessageAsStringBuilder.ToString();
        }
```

Why is this the case? Because the String object is immutable. This means that a new String object has to be allocated on the heap every time a concatenation occurs.

Tuning

This is a very brief section on tuning tools. For specific details on tuning and profiling, numerous relevant text can be found on the internet.

Timing Code Segments

The .Net StopWatch object can provide timings to an approximate resolution of a microsecond. This is useful for timing sections of code when reworking them during an optimization process.

MSIL

When performance is critical it's always good to take time to check the MSIL (Microsoft Intermediate Language) code that is generated. This can highlight when a particular piece of code may benefit from a simple re-write to optimize it further.

Profiling Code

Use Visual Studio code analysis tools for code profiling. This should be the first port of call to determine the areas of code that will benefit most from any optimizations.

Java Performance – Avoiding Garbage Collection

By Mahesh Kumaraguru

The Garbage Collector

Java's automatic memory management has its advantages and disadvantages.

Advantages	Disadvantages
There are no situations of invalid memory pointers or dangling references because Java garbage collects objects that do not have any active references.	The Garbage Collector pauses program when doing garbage collection. The program cannot control unused Object deletion.
The program does not have to bother about size of Object when allocating memory.	The program cannot allocate a large number of objects in a single operation by allocating a large enough buffer.

Object Pools

One widely used and popular method to avoid garbage collection is by using Object pooling. Object pooling is useful when Objects are expensive to create and Objects are short lived, e.g. new Objects which are created are not required till the end of an application, e.g. their life is over before the program ends and they can either be garbage collected by the JVM or recycled by the program. In Object pooling, the program takes responsibility for recycling memory instead of Java recycling memory.

For purposes of this article, let’s call the class which implements Object pooling ObjectPool.java which manages instances of a class called MyObject.java

ObjectPool.java

```
Public class ObjectPool
{

       public static final  int initSize = 10;      //assign this value based on your
                                                   //requirements of initial pool size.

       private final ArrayList objectsInUse;
       private final ArrayList freeObjects;

       private static ObjectPool instance = new ObjectPool();
       public static ObjectPool getInstance()
       {
              if(instance = = null)
              {
```

ObjectPool.java

```
                        instance = new ObjectPool();
                }
                return instance;
        }

        private ObjectPool()
        {
                freeObjects = new ArrayList(initSize);
                for(int i = 0; i < initSize; i++)
                {
                        freeObjects.add(new MyObject());
                }

                objectsInUse = new ArrayList(initSize);
        }

        public synchronized MyObject getNew(String requestor)
        {

                if(freeObjects.size() > 0)
                {
                        MyObject retVal = freeObjects.remove(0);
                        objectsInUse.add(retVal);
                        retVal.setRequestor(requestor);
                        return retVal;
                }
                else
                {
                        MyObject newObj = new MyObject();
                        objectsInUse.add(newObj);
                        newObj.setRequestor(requestor);
                        return newObj;
                }
        }

        public synchronized void release(MyObject o)
        {
                o.reset();
                objectsInUse.remove(o);
                freeObjects.add(o);
        }

        public void finalize()
        {
                System.out.println("Starting Finalize ObjectPool");
                freeObjects.trimToSize();
                System.out.println("Number of Free Objects = " + freeObjects.size());
                objectsInUse.trimToSize();
                System.out.println("Number of Objects in Use = " + objectsInUse.size());
                System.out.println("Objects in Use details");
                for(int i = 0; i < objectsInUse.size(); i++)
                {
                        System.out.println("Requestor [" + i + "] " +
objectsInUse.getRequestor());
                }
                System.out.println("Ending Finalize ObjectPool");
        }


}
```

When a program needs an instance of MyObject, instead of calling new MyObject() to create a new instance of the MyObject, it calls the getNew() method of the Object pool. If the ObjectPool has unused MyObject instances available, it returns one of them. Otherwise, it creates a new MyObject and returns it.

When the program is done with using the instance of MyObject, instead of the Garbage Collector destroying the Object, the Object is returned to the pool by calling the release method, and the ObjectPool makes it available for reuse.

For instances of MyObject class to be reusable by pooling, it must adhere to the following rules:

- MyObject must provide a reset method, which resets the state of MyObject to an initial state, e.g. removes all state that the user of ObjectPool.java stored into the instance.
- MyObject must not use a constructor to populate instance variables because constructor is only called when new is used.
- All values must be populated using methods; good Java coding conventions would require you to call these methods, e.g. setSomeValue(SomeValue sv), or the values must be declared public. An advantage of using setter method is that the methods can do validation before setting the value.

A String identification of the class requesting a new Object instance by calling the getNew method is useful for debugging memory leaks. When the application is being shut down, the finalize method of ObjectPool is called which prints out requestors of Objects which were not released back to the pool.

Resilience

By Cooper Noone, Counterparty Systems

Introduction

This article covers various approaches to help reduce or eliminate interruptions in service by offering ideas for implementation options that allow FIX messages to flow between counterparties continuously and unbroken, regardless of what adversity the operating environment encounters.

Resilience is the ability of software to gracefully handle and recover from events that impact its ability to perform under normal operating conditions. Clients' expectations on performance, availability and reliability of enterprise systems have grown tremendously in recent years. To prevent potential problems with data integrity or availability, the industry and many customers and risk management organizations have made stronger resilience as high a priority as business requirements. It's no wonder with the amount of capital at risk during the trading day, any interruption of service puts the firm at potential financial risk.

It is essential for technology teams to proactively discuss resiliency with customers or management to safeguard against data loss and/or service interruptions down the road. Based on my experience, even the most robust systems' implementations encounter the unexpected, making it less a matter of *if* an interruption in service will happen, but *when*. A well-designed architecture will reduce or eliminate operational risk with an industrial-strength resiliency plan.

Levels of Resilience

When discussing resilience, there are multiple platforms and service-levels that should be identified and addressed by the developer. Especially for inter-company communication like FIX, it is important for the developer to understand all of the factors that could potentially impact FIX message traffic when introducing a more complex design to address resilience. Typically, FIX implementations are dependent on platforms that include servers, software, and networks. Each of these platforms can be implemented with varying service-levels of resilience which are sometimes referred to as fault-tolerant, highly-available, and disaster recovery.

Fault-tolerant (FT) systems provide the highest level of resiliency where zero downtime is required. These systems are designed to provide 100% up-time with no interruption in service during the specified service-level agreement (for example, during market hours). Highly-available (HA) systems typically offer a service-level that involves a minimal amount of downtime (for example, 0-5 minutes). Disaster recovery (DR) service-levels are typically longer than HA service-levels (for example, 1-2 hours) because the event that triggered a DR scenario is catastrophic outage impacting a computer room, floor, building, or city that affects a broad range of systems.

Ensuring resilience is challenging at each level. An implementation solution is only as good as the level of resilience of the other platforms. For example, you can implement the most robust fault-tolerant application design possible, but if your network or server hardware and software are not fault-tolerant as well, then your application is held to the mercy of the other levels of resilience.

The majority of FIX engine implementations in existence today establish TCP/IP socket connections between each other (which is no longer required with FIX 5.0 that decoupled the FIX session layer from the FIX transport layer); as a result, they cannot be made truly fault-tolerant. If a FIX engine that uses sockets upends unexpectedly, at a minimum there has to be a brief interruption of messaging while the socket connection is reestablished. Therefore, it is desirable to design the FIX engine and supporting processes to handle faults so that the software can recover quickly and with little or no impact to client processes.

For the purposes of this article, I will limit the scope of discussion to implementations involving FIX engine redundancy. Fully transparent fault-tolerant application, network and operating system level redundancy can be assumed for the purposes of this discussion.

Persistence

Persistence is the ability of a FIX engine to store and recall information that will allow a system to return to a known state after an outage or unexpected system failure. Ideally, an OMS (or EMS) wants to dependably send and receive messages with counterparties without worrying whether or not the FIX engine is up and running. The OMS just wants to connect to a FIX engine and start giving/getting data ignorant of how FIX engine persistence is handled behind the scenes. But what type of persistence should a FIX engine implement, if any?

Choosing a persistence approach that is best for your implementation will be driven by your requirements for latency, data integrity, service level agreements, time-to-market, and cost. Low latency implementations (typically used by brokers and exchanges) sacrifice quick recovery time by either choosing persistence methods that have little or no impact on the speed of data flow, or by eliminating persistence altogether. When an implementation defines system availability as paramount, then persistence methods that provide fast recovery time win out at the cost of slower data delivery.

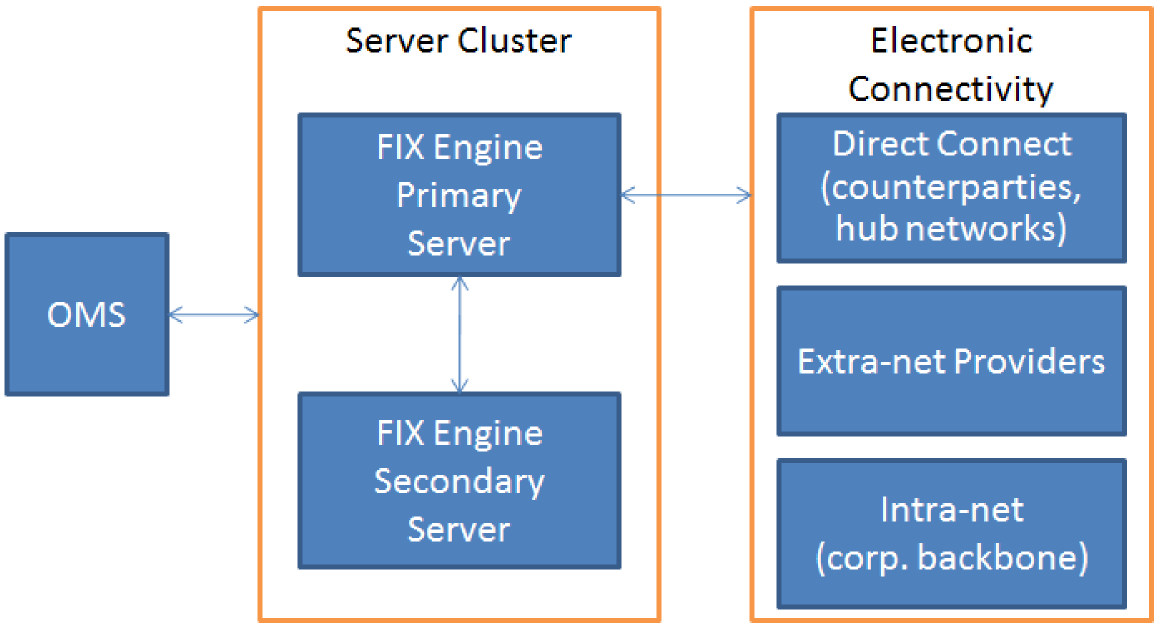
Using FIX sequence numbers to implement persistence is a straightforward and lightweight approach. For example, if a FIX engine is bounced and needs to pick up message handling from where it left off with each counterparty, then it could issue a ResendRequest (MsgType=2, BeginSeqNo=1, EndSeqNum=0) message for each session. Alternatively, the FIX engine could ask the application (OMS) what was the last message it received so that it could use the correct sequence number on its counterparty connections.

Fortunately, FIX engines have evolved and robust persistence is, and should be, a key feature of many commercially available vendor products on the market. If yours doesn't have some type of persistence capability, it may be time to start asking your vendor why not, or start looking around at other vendors that do have persistence implemented if you are following a buy (vs. build) strategy.

Mirrored & Clustered Servers

Advanced persistence models involve one primary server and at least one or more backup or mirrored servers in a cluster. All FIX engines in a server pair or cluster are running simultaneously, but only one engine is designated as primary. The primary engine is responsible for establishing connections and maintaining sessions with counterparties, sending and receiving messages to the OMS, and to keep the other engines in the cluster in sync with current state.

The non-primary engines are responsible for keeping up with the state of the primary engine, monitoring the primary engine, and to assume the role of primary engine (based on chain of command) in the event of a failure. Once a failure of the primary engine occurs, due to either hardware or software, the secondary engine should inform other engines in the cluster, if any, of its intent to become primary, take over as primary, and then reestablish connections with the OMS and counterparties and resume session traffic.


Server Cluster
Electronic Connectivity
FIX Engine Primary Server
FIX Engine Secondary Server
OMS
Direct Connect (counterparties, hub networks)
Extra-net Providers
Intra-net (corp. backbone)

Basic Local Persistence Model

In the basic persistence model presented above, the OMS connects to a virtual IP address that is an alias which points to the primary FIX engine server. The primary FIX engine is connected via sockets to the external FIX sources/destinations.

In the event of a software or hardware failure that impacts the primary FIX engine, the secondary FIX engine becomes primary and reconnects to counterparties, and the OMS reconnects to the FIX engine via the same IP address. Once the problem with the faulty server is fixed, it can be re-introduced into the server cluster, synchronized with the new primary server, and then becomes the new secondary server (depending on the number of servers in the cluster).

In the event of a software or hardware failure that impacts the secondary FIX engine, the primary FIX engine can optionally continue network persistence with another server in the cluster, switch to file persistence, or continue processing messages without a synchronized backup server, obviously the least desirable and most risky option.

Persisting data using synchronous communication will help ensure a cluster of FIX servers will maintain accurate state and data more reliably in the event of an outage. The associated cost is that significant latency is introduced to guarantee this data integrity. For example, when the primary server sends each message it gets to the secondary server, the secondary server acknowledges receipt before the primary server continues processing messages. In asynchronous communication, the primary server would send the secondary server a message and continue processing messages without waiting for a response from the secondary server. There is less assurance the two servers are in sync, but less latency is introduced in processing messages.

There are a few mechanisms available to consider for implementing persistence between servers, each with various trade-offs. For example:

- TCP/IP sockets or multi-casting over a LAN or WAN
- Flat file
- Database

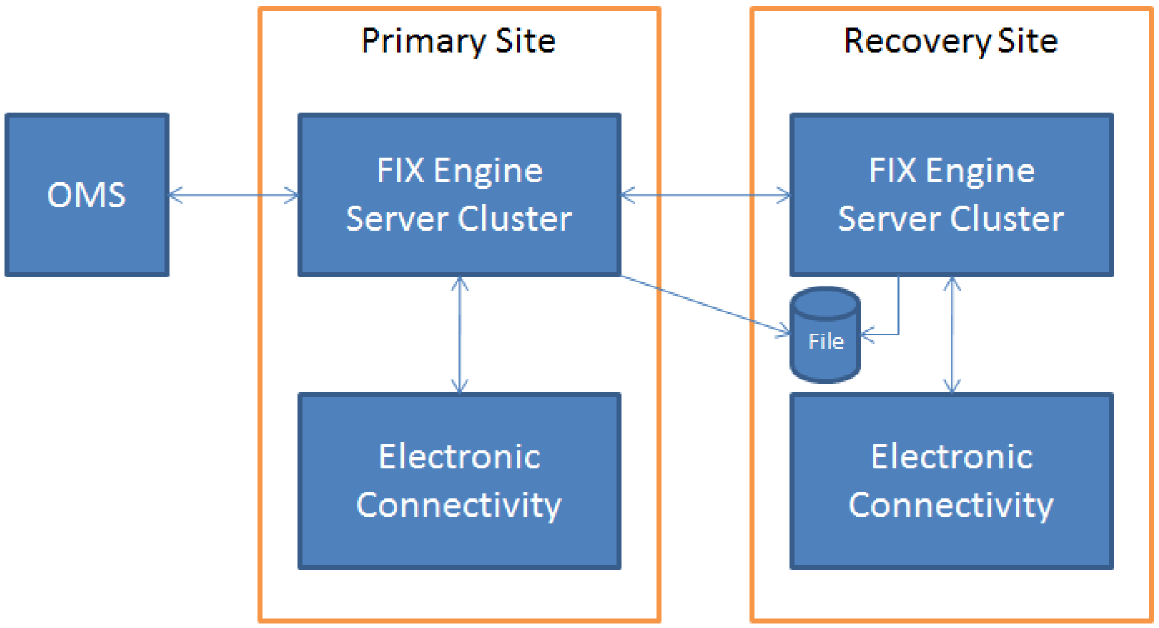
Consider that it takes roughly 6 times longer to commit a transaction to disk for file persistence than a transaction to be transmitted by LAN for network persistence[1]. Persistence over a WAN to remote sites can be more challenging due to network latency and bandwidth limitations that vary from firm to firm—in addition, distance between sites adds latency due to physical limitations on the speed at which data/light travels over network lines. The overhead involved with persisting to a database adds even more time to the transaction than file persistence, but there is added convenience by having structured reference data available that can be accessed easily in various ways.

Creating a persistent data store in a flat file on a shared file system is a flexible and somewhat more resilient approach than network persistence because FIX engines are able to recover quickly after network failures. In addition, FIX engines in remote sites can access the information in the flat file in the event of a disaster. It is important to ensure the data store is secured against unauthorized access due to the nature of the sensitive data contained within.

Disaster Recovery

Disaster recovery service-levels can vary greatly from firm to firm, depending on the level of overall commitment by senior management to a DR strategy. Even having the smallest system implemented with DR requires a significant investment in supporting systems, testing, and training. As mentioned earlier, a catastrophic event that triggers a DR scenario can impact a broad range of systems. Switching systems to run from a recovery site instead of the primary site can involve cold starting servers and/or software, intra-day modifications to the operating environment, and resynchronization of data to get systems back to a running state.

[1] Orc Software, CameronFIX support documentation



Basic Disaster Recovery Model

The basic DR design model (see diagram) is a cold or warm start model that has two FIX engine clusters, one at the primary site and one at the recovery site. Each site has independent internal server and network infrastructures, as well as a separate infrastructure for external connectivity. The server clusters stay in sync via asynchronous network communication or flat file persistence. The cluster in the recovery site in cold start mode can load the flat file data at start up, or synchronize FIX sequence numbers with counterparties. Warm start mode has the cluster up and running as backup servers.

Key characteristics of this design are:

- Cost to maintain – since the cold servers are not part of daily production processing, separate DR testing is needed to ensure they will be ready when needed
- Recovery time during a DR event – additional time is needed to get the servers up and into their running state to take over as primary servers
- Risk of data loss between sites – there is a chance that not all of the data makes it to the recovery site when the primary server cluster fails

Factors that influence technology architects when designing a DR solution for FIX engine servers include geographic distance between servers, network utilization and bandwidth, latency, system/customer requirements, risk tolerance, and cost. Bandwidth costs have come down in recent years, allowing some DR solutions to play a dual-role as FT/HA backup servers and DR servers simultaneously. Employing a combined HA/DR solution will allow for greater flexibility for load balancing over a larger range of servers giving greater scalability and data security to the FIX infrastructure. However, low latency is critical for some firms, thus putting limitations on the geographic distance betweens redundant server clusters.

Key factors to consider when designing a combined FT/HA/DR solution are:

- High bandwidth between sites is needed due to the larger amount of data going across the network that is extremely time critical; at least double the amount of the highest amount of network traffic to all counterparties, so that it can carry FIX traffic and persistence traffic.
- Since all servers are active participants in the cluster, the amount of DR testing is reduced
- Recovery time during a DR event is eliminated – regardless of where other downstream or upstream systems reside, the FIX server cluster is always available
- Less risk of data loss between sites – since data is kept in sync in real-time with all the servers in the cluster, this risk is minimized

Conclusion

In this article we've covered resilience and persistence at a high-level. There are many possible solutions that can be implemented depending on your own unique requirements. The good news is that there is help available to guide you through the many architectural choices. With the many of the hundreds of FPL member firms having already addressed this issue at some degree, there are a vast amount of resources to tap into to help send you in the right direction.

FIX Java Type Safety

By Mahesh Kumaraguru

High level object oriented languages like Java, C++, C# etc. permit user defined types which promotes type safety. The cost of type safety in using classes instead of primitive types is justified by the robustness of the program, and when used properly does not degrade performance.

For example, in the FIX Protocol, message types (permitted values of Tag 35) are Strings. The programmer has two choices:

1. Using Strings (non typesafe primitive datatype) to represent permitted values of Tag 35.
2. Using a class to represent permitted values of Tag 35.

MessageType Using non-typesafe String Primitive Datatype

Definition

```
public interface MessageType
{

        String LOGON            = "A";
        String HEARTBEAT        = "0";
        String ORDER_SINGLE     = "D";
        String EXECUTION        = "8";
}
```

A method which uses MessageType as Parameter

```
returnTypeObject aMethod(String mt) throws IllegalArgumentException
{
        if(!mt.equals(MessageType.LOGON)              |
                !mt.equals(MessageType.HEARTBEAT)     |
                !mt.equals(MessageType.ORDER_SINGLE)  |
                !mt.equals(MessageType.EXECUTION))
        {
                throw new IllegalArgumentException(mt);
        }



/********************************************
Validation of mt is required because any String value can be passed
Comparison has to be performed by calling equals method of String class, == comparison
cannot be used.
********************************************/


}
```

Calling aMethod

```
String invalidMessageType = "I";
//No compile Error, any String can be used as Message Type
```

```
try
{
        aMethod(invalidMessageType);
}
catch(IllegalArgumentException iae)
{

}
//Exception handling code needs to be present
```

Disadvantages of using non-type safe primitive datatypes:

1. Comparisons need to be performed using the equals method, thus incurring a method call.
2. Errors can only be detected by testing because errors cannot be detected at compile time.
3. Increased need to validate values because invalid values can be used.
4. Incorrect assignments would compile, for example:
   ```
   String ordStat;
   String exTyp;
   aMethod(String os, String et)  // os is for ordStat, et is for exTyp
   {
      ordStat = et;  // no compile error, though os should have been used here
      exTyp  = os;   // no compile error, though et should have been used here
   }
   ```
   Tedious code reviews / exhaustive testing is required to catch these types of errors.
5. Method overloading cannot be used when primitive types are used. For example,
   ```
   set(String et)      //et is ExecType
   set(String os)      //os is OrderStatus
   ```
   cannot be used; it would give a compile error because both of these method signatures are same.

TypeSafe MessageType Using a Multiton Class

Definition

```
Public final class MessageType
{
        public final String value;
        private MessageType(String s)
        {
                value = s;
        }

public static final MessageType LOGON = new MessageType(“A”);
public static final MessageType HEARTBEAT = new MessageType(“0”);
public static final MessageType ORDER_SINGLE = new MessageType(“D”);
public static final MessageType EXECUTION = new MessageType(“8”);
}
```

A method which uses MessageType as Parameter

```
returnTypeObject aMethod(MessageType mt)
{
```

```
/******************************
No need to validate because all MessageType values are strictly defined at Compile time
******************************/
}
```

Example :- Calling aMethod using an Invalid MessageType

```
MessageType invalidMessageType = new MessageType("I");
aMethod(invalidMessageType);

/*********************************************
First line above would give a compile error because
MessageType constructor is private, so an invalidMessageType
cannot be created. Only the correct argument of type
MessageType can be passed to aMethod using the
public static final values defined in MessageType class
*********************************************/
```

Example :- Calling aMethod using a Valid MessageType

```
aMethod(MessageType.ORDER_SINGLE);

/*********************************************
Only valid message types of type MessageType can be passed can be passed to aMethod
using the public static final values defined in MessageType class
*********************************************/
```

A more advanced example would be to include FIX Version as a parameter in MessageType to indicate the FIX Version in which the MessageType was introduced.

Definition of FIXVersion

```
Public final class FIXVersion
{
       public final int       majorVersion;
       public final int       minorVersion;
       public final String    beginString;
       private FIXVersion(int mjv, int mnv, String bs)
       {
              majorVersion = mjv;
              minorVersion = mnv;
              beginString = bs;
       }

       public String toString()
       {
              return beginString;
       }

public static final FIXVersion FIX40 = new
                      FIXVersion(4,0, "FIX.4.0");
public static final FIXVersion FIX41 = new
                      FIXVersion(4,1, "FIX.4.1");
public static final FIXVersion FIX42 = new
                      FIXVersion(4,2, "FIX.4.2");
public static final FIXVersion FIX43 = new
                      FIXVersion(4,3, "FIX.4.3");
public static final FIXVersion FIX44 = new
                      FIXVersion(4,4, "FIX.4.4");
public static final FIXVersion FIX50 = new
                      FIXVersion(5,0, "FIXT.1.1");
public boolean isGreaterOrEqual(FIXVersion inFV)
{
        if(this.majorVersion < inFV.majorVersion) return false;

        else if((this.majorVersion == inFV.majorVersion) &
                (this.minorVersion < inFV.minorVersion))
                        return false;

        return true;
}

//add other comparison methods

}
```

Definition of MessageType

```
Public final class MessageType
{
        public final String value;
        public final FIXVersion introducedInVersion;
        private MessageType(String s, FIXVersion iiv)
        {
                value = s;
                introducedInVersion = iiv;
        }

public static final MessageType LOGON        =
               new MessageType("A", FIXVersion.FIX40);
public static final MessageType HEARTBEAT    =
               new MessageType("0", FIXVersion.FIX40);
public static final MessageType ORDER_SINGLE =
               new MessageType("D", FIXVersion.FIX40);
public static final MessageType EXECUTION    =
               new MessageType("8", FIXVersion.FIX40);
public static final MessageType ORDER_MULTILEG =
               new MessageType("AB", FIXVersion.FIX43);
}
```

A method which uses MessageType and associated FIXVersion as Parameter

```
Class someHelperClass
{
        FIXVersion thisVersion; //initialized in constructor
returnTypeObject aMethod(MessageType mt)
{
        //Processing based on FIXVersion can be done
        if(thisVersion.isGreaterOrEqual(mt.introducedInVersion))
        {

        }
        else
        {
                if(mt.introducedInVersion == FIXVersion.FIX44)
                {
                }
                else
                {
                }
        }
}
}
```

The above simple example of MessageType represents the concept of promoting type safety inherently useful when developing programs in high level object oriented languages like Java, C++ etc. This example can be extended to include FIXTag, ExecTransType, ExecType, OrderStatus, OrderType, TimeInForce, TradeSide, SecurityIDSource etc., e,g, all fields of FIX could be defined.

Advantages of TypeSafe code:

1. Comparisons can be performed using = = which has the same performance as comparing primitives.
2. Much faster development and much more robust code which can detect lots of errors at compile time rather than at run time, thus reducing development and testing effort.
3. Reduces need to validate values because invalid values cannot be created / used.
4. Incorrect assignments would not compile when type safe code is used, e.g.
   ```
   OrderStatus ordStat;
   ExecType exTyp;
   aMethod(OrderStatus os, ExecType et)
   {
      ordStat = et;//compile error : cannot convert from ExecType to OrderStatus
      exTyp  = os;//compile error : cannot convert from OrderStatus to ExecType
   }
   ```
5. Another advantage of using types is that method overloading can be used, making code more readable. For example:
   ```
   set(ExecType et)
   set(OrderStatus os)
   ```
   where ExecType and OrderStatus are multiton classes defining valid values of ExecType and OrderStatus respectively would compile without errors.

Virtually all constant values can be coded type safe where complex validations and interactions between tag values, message types and FIX Versions can be represented flawlessly – the Object Oriented advantage.

Disabling The Nagle Algorithm

By John Prewett, Citi

The Problem

The Nagle algorithm is a means of improving the efficiency of an IP network by reducing the number of packets that need to be sent over a TCP session. In very general terms, it does this by combining multiple messages from an application into a single packet. In order to achieve this, the Nagle algorithm **delays** small messages, hoping a subsequent message will be sent, so that the two messages can be combined into a single packet. In the attempt to achieve this objective, a worst-case scenario is that **a message can be delayed by as much as 500 milliseconds.**

For a more precise explanation of the Nagle algorithm, I recommend reading a Wikipedia entry on the subject: http://en.wikipedia.org/wiki/Nagle%27s_algorithm

The Delayed Acknowledgement algorithm is another means of improving the efficiency of an IP network by reducing the number of packets that need to be sent. Again in general terms, it delays sending TCP acknowledgements, hoping that it can piggy-back them on an outgoing message.

Both Nagle & Delayed Acknowledgement algorithms are built into almost all TCP implementations as part of the operating system. The Nagle algorithm is turned on by default on all TCP sessions when they are established. The Delayed Ack algorithm is turned on by default for all TCP sessions using a particular network adaptor.

Most FIX engines use TCP as their underlying transport.

If a FIX engine is using a TCP session with the Nagle algorithm enabled and it sends two messages in quick succession to a destination which has the Delayed Ack algorithm enabled, the second message can be delayed by as much as 500 milliseconds.

This unfortunate interaction between these two algorithms is well documented in many places on the Internet. Here is a link for more in-depth reading:
http://www.stuartcheshire.org/papers/NagleDelayedAck/

In these days of microsecond latencies, experiencing an occasional message delay of up to 500 milliseconds is nothing short of catastrophic.

Determining if you are Suffering from the Problem

If a FIX engine is using a TCP session with the Nagle algorithm enabled and it sends two messages in quick succession to a destination which has the Delayed Ack algorithm enabled, the second message can be delayed by as much as 500 milliseconds.

Start up a FIX acceptor and a FIX initiator and connect both ends of the FIX session together. Ensure both initiator & acceptor log the time messages are received with timestamps accurate to milliseconds.

Send two messages from the FIX initiator with the second message being sent immediately after the first message. If the receiving timestamps at the FIX acceptor are more that a “reasonable” amount apart (depends on how close together you sent the two messages), you are potentially suffering from the problem on the FIX initiator.

Now repeat the procedure, except this time send the two messages from the FIX acceptor to the FIX initiator. Again, if the receiving timestamps at the FIX initiator are more that a “reasonable” amount apart (depends on how close together you sent the two messages), you are potentially suffering from the problem on the FIX acceptor.

Note that this problem can be detected without any usage of the FIX SendingTime tag. Comparing SendingTime from a message with the time that message is received is potentially misleading if the clocks on the FIX initiatior & FIX acceptor are out of synchronization.

Apart from this simple approach, it is also recommended to use a network sniffer. It is always a good idea to look at the TCP packets. Behavior may not be what you believe or expect. It can be misleading to sniff the packets on the machine where your FIX engine resides. It is much better practice to sniff network traffic at the last piece of hardware before packets leave your datacenter. This will catch badly behaving network appliances such as firewalls, load-balancers, etc.

Usage of a sniffer managed to detect the fact that a certain load-balancer was actually terminating the TCP session from the FIX engine at the load-balancer and establishing a completely new TCP session to the customer, acting as an application-level message router. While Nagle was disabled on the TCP session from FIX engine to load-balancer, the TCP session from load-balancer to the customer had Nagle enabled, and thus major latencies were occurring.

The Solution

There are two possible solutions to the problem.

1. Disable the Nagle algorithm at both ends of the FIX (and TCP) session.
   This solution results in both ends being able to send without any delays, subject to the available TCP window size at the receiver.
2. Disable both the Nagle and Delayed Ack algorithms at one end of the FIX (and TCP) session.
   This solution results in almost the same results as solution #1 except for the fact that there will be more packets transmitted because TCP Acks at one end will not piggy-back on outgoing messages.

The preferable solution is #1 above. Solution #2 should only be chosen if solution #1 is not possible.

[Rhetorical Question] Why would solution #1 not be possible?

[Answer] Most FIX sessions are established between two corporations. You may well have sufficient control over *your* end of the FIX session, but not the other end. If the other end of the FIX session doesn't have the capability to disable Nagle, then it is possible that solution #2 is your only alternative.

The downside to both solutions is that there will be more packets transmitted over the network. This in turn will result in greater bandwidth utilization although it is anticipated that the percentage increase should be relatively insignificant. Solution #2 results in more packets than solution #1.

Another downside to solution #2 is that disabling the Delayed Ack algorithm is done on the level of the network adaptor. Thus all TCP sessions using that adaptor will be affected by this setting, not just the ones you might want.

Implementing the Solution

The way to disable the Nagle algorithm varies with operating system and programming language. Most TCP implementations offer some type of sockets interface.

To disable the Nagle algorithm in C/C++ on a generic sockets interface looks like this:

```
int value = 1;                        /* Option value            */
int result = setsockopt(mySocket,     /* Socket to be affected   */
                        IPPROTO_TCP,  /* TCP option              */
                        TCP_NODELAY,  /* Option name             */
                        (char *)&value, /* Option value to be set  */
                        sizeof(int));   /* Size of option value    */
if (result < 0)
   ... didn't work, deal with the error ...
```

If you are establishing the TCP session (i.e. your end is the FIX initiator), disable the Nagle algorithm immediately after creating the socket.

If you are a FIX acceptor, disable the Nagle algorithm on the listening socket. Additional sockets created when the other end establishes the session will automatically inherit this setting.

The method to disable the Delayed Ack algorithm varies wildly from operating system to operating system. You need to configure the TCP layer of the network adaptor not to use Delayed Ack. Typically, this isn't done programmatically. It is done through re-configuring the operating system.

I/O Models and I/O Completion Ports

By Qingjun Wei, Teraspaces

The Problem

TCP is the de-facto transport layer of the FIX protocol. This paper aims to introduce different design patterns of TCP I/O and to discuss how those patterns are applied to the FIX transport layer.

Operation System Native TCP I/O

The good old `select()` function can operate on more than one sockets. It wait for status of one or more sockets, waiting if necessary, to perform synchronous I/O.

On single socket, there are three other different kinds of native I/O patterns provided by modern operating systems, including Windows and Linux/UNIX:

1. Blocking I/O: This would translate to a `read()/write()` on a blocking socket. The thread that makes such calls would block until there was some data available to read/write or the socket was closed.
2. Non-blocking, synchronous I/O: This would translate to a `read()/write()` on a non-blocking socket. The thread that makes such calls would return immediately, either with the data read/written, or with a return code of error and an error code that indicates that the IO operation could not complete, or socket was closed. On the Windows platform, the return code will be `SOCKET_ERROR` and error code return from `WSAGetLastError()` will be `WSAEWOULDBLOCK`. On a Unix system, the return code is -1 and a call to `errno()` will return `EWOULDBLOCK`.
3. Non-blocking, asynchronous I/O: This would translate to the Windows overlapped I/O extension with WSA* functions such as `WSASend()/WSARecv()` (Microsoft), Unix SIGIO mechanisms or POSIX aio_* functions. Essentially, these IO calls return immediately, and the OS starts doing the operation in a separate (kernel level) thread; when the operation is ready, the user code is given some notification.

Overhead Analysis of I/O Related Operations

Every instruction in your code results in a certain overhead on your CPU. We list some types of overhead associated with I/O related operations below:

1. Thread Creation
   Thread creation is still very expensive in modern operation systems. It not only consumes a lot of CPU time, but also incurs a huge latency. The latency of

creating a thread on both Windows and UNIX is measured at the millisecond level, even though the performance of thread creation on UNIX is significantly better than on Windows (Zabatta and Ying).

2. Context Switch
   A context switch is the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource. Typically the cost of a context switch is about 4000-120000 CPU cycles, depending on whether the working set is in cache.
   I/O related context could be caused by the following reasons:
   a. A blocking I/O call
      When a thread is waiting for a blocking I/O, a context switch is triggered to yield the CPU to another thread.
      When the blocking I/O is completed, another context switch is triggered to give the CPU to the blocked thread.
   b. A blocking synchronization operation
      When a thread is waiting for a synchronization object, or has acquired control of it from the waiting mode, a context switch is triggered. Although a synchronization operation is not directly related with I/O, it is often associated with all kinds of I/O operations. For example, an I/O operation may trigger a synchronization operation. In practice, when a FIX message is received, a thread may be triggered to process the message by signaling a synchronization object. Another example is non-blocking, asynchronous I/O. When the I/O is completed, a callback function may be called, which may in turn trigger a synchronization operation. In process synchronization (such as a critical section) is more efficient on Windows than on Linux, while out of process synchronization is more efficient on Linux (Zabatta and Ying).
3. Memory
   Memory overhead is not particularly associated with TCP I/O operations. However, each thread does incur memory overhead. Each thread has to be allocated a thread local stack. On the Windows platform, the default thread stack size is 1 MB (Microsoft). Although the initially committed pages do not utilize physical memory until they are referenced, it does take up virtual address space. On a 32-bit machine, the total usable address space is only 2-3.5 GB. Modern Operating Systems can support 3 or 3.5 GB of user address space on 32-bit machines. A certain compiler option may be required to compile code that can utilize the address space over 2GB. This translates to less than 3000 threads, which is the upper limit of the number of threads allowed. In practice, the number should be much lower than 3000.

I/O Completion Ports

As we mentioned previously, both I/O operation and thread scheduling may cause a context switch. By default, asynchronous I/O callbacks are associated with a system thread pool. It will be optimal to have a fine-tuned control over the maximum number of threads in the thread pool used in asynchronous I/O calls. In order to achieve it, Windows introduced the concept of an I/O completion port (Microsoft).

I/O completion ports bring additional efficiency by giving the programmer fine-tuned control over the number of threads in the threading pool associated with a group of I/O handles. Ideally the number of threads should not be less than the number of CPU cores (or hyper-threads). After a callback is performed, if another I/O operation in the group is completed and queued, the current pooled thread can immediately perform another callback without the need for an additional context switch.

Overhead Analysis on TCP-Based FIX Implementation

FIX protocol has its own characteristics that differs itself from other communication protocols:

- FIX protocol is stateful
  The FIX protocol maintains a sequence number for messages and defines a recovery mechanism in case of unexpected loss of connection of a FIX session. Typically, the FIX session is transported through a TCP channel that stays connected for hours or even for days. Unlike the HTTP protocol, which is stateless and clients frequently connect and disconnect during transactions, FIX session has a different performance profile. For an HTTP session, because it constantly gets connected and disconnected, it is absolutely impractical to assign one thread to each connection and perform blocking I/O on it. The overhead of thread creation is prohibitive. A FIX session stays connected for a much longer time, which makes the performance impact of thread creation negligible.
- Limits on Number of FIX Sessions
  The FIX protocol is designed to interconnect among financial institutions. A large firm that provides FIX connectivity to clients may have over 1000 FIX sessions on a single server. If each FIX session is associated to 1 or 2 threads, in a 32-bit machine, 1000 is a lot and it may reach the virtual address limit. However, on a 64-bit machine, having tens of thousands of concurrent threads can be achieved. At any rate, a FIX server is different from some other services such as an instant messaging server, which may have hundreds of thousands of concurrent TCP connections in order to serve customers from all over the world. Another problem associated with threading is the context switch. Having more threads in the system and associating each thread to a TCP channel will certainly increase the number of context switches linearly.

Language Mapping

Both Java and .Net provide built-in classes for all three types of native I/O operations mentioned above.

I/O Design Patterns

Some people have successfully used the select() function to achieve good performance on a system with large amount of FIX sessions, subject to the limitations of FD_SETSIZE. It is possible because select() function may return with status change of more than one connections, especially when the connection pool is large. The actual I/O calls will trigger no context switch. In this case select() can be used as a single thread job dispatcher.

On single socket, there are two popular I/O system design patterns, the Reactor Pattern and the Proactor Pattern.

According to Libman and Gilbourd, the Reactor pattern, which usually would use synchronous I/O (either blocking or non-blocking), would work like this (read operation is used as an example, write will work in similar way):

- An event handler declares interest in I/O events that indicate readiness for read on a particular socket.
- The event de-multiplexer waits for events.
- An event comes in and wakes-up the de-multiplexor, and the de-multiplexor calls the appropriate handler.
- The event handler performs the actual read operation, handles the data read, declares renewed interest in I/O events, and returns control to the dispatcher.

The Proactor pattern, as true asynchronous I/O, would work like this:

- A handler initiates an asynchronous read operation (note: the OS must support asynchronous I/O). In this case, the handler does not care about I/O readiness events, but instead registers interest in receiving completion events.
- The event de-multiplexor waits until the operation is completed.
- While the event de-multiplexor waits, the OS executes the read operation in a parallel kernel thread, puts data into a user-defined buffer, and notifies the event de-multiplexor that the read is complete.
- The event de-multiplexor calls the appropriate handler.
- The event handler handles the data from user defined buffer, starts a new asynchronous operation, and returns control to the event de-multiplexor.

Both design patterns are language neutral, i.e. they can be implemented in virtually any modern programming language.

Implementation Issues

The Reactor pattern is the easiest to implement. It depends on the system kernel to deal with thread scheduling and context switches. The control flow of the algorithm is clear and natural. This simplicity and low code maintenance cost is achieved at some additional cost of performance overhead.

The Proactor pattern can achieve the highest possible efficiency, if designed and implemented right. The idea is to partition the work flow into several small pieces. Each piece is a small chunk of a task. The boundaries of the each task are what normally cause context switches in a "normal" control flow.

A thread pool and one or more queues of tasks should be maintained. Your own scheduler should be built to schedule the tasks in the queue among threads in the thread pool. The scheduler is basically a state machine. Part of the job is to emulate the kernel's scheduling of threads and context switches.

The scheduler and each partitioned task should not contain additional operations that may possibly cause a context switch (which will defeat the purpose) or blocking (which is the wrong implementation), unless the operation is used to manipulate the thread pool.

Conclusion

In this paper we discussed the available I/O technology and design patterns and how they may possibly affect the performance of FIX Protocol based systems. System designers should thoroughly consider the cost/efficiency analysis of all factors to decide which technology is the best fit for the nature of your business.

Bibliography

Libman, Alexander and Vladimir Gilbourd. "Comparing Two High-Performance I/O Design Patterns." 25 November 2005. Artima Developer. <http://www.artima.com/articles/io_design_patterns.html>.

Microsoft. "Synchronous and Asynchronous I/O." Microsoft Developer Network. <http://msdn.microsoft.com/en-us/library/aa365683(VS.85).aspx>.

"Thread Stack Size." 05 02 2009. Microsoft Developer Network. <http://msdn.microsoft.com/en-us/library/ms686774(VS.85).aspx>.

Zabatta, Fabian and Kevin Ying. "A Thread Performance Comparison:Windows NT and Solaris on A Symmetric Multiprocessor." 3-5 August 1998. 2nd USENIX Windows NT Symposium Pp. 57–66 of the Proceedings. <http://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/zabatta/zabatta_html/zabatta.html>.